DOCUMENT RESUME

ED 076 213 LI 004 318

AUTHOR Hemming, Cliff; Smith R. J., II
TITLE An Introduction to Register Transfer Level Simulation of Digital Systems.
INSTITUTION Southern Methodist Univ., Dallas, Tex. Computer Science/Operations Research Center.
REPORT NO TR-CP-73013
PUB DATE May 73
NOTE 38p.; (32 References)

EDRS PRICE MF-$0.65 HC-$3.29
DESCRIPTORS *Computer Programs; Computers; *Computer Science; *Digital Computers; Program Descriptions; Program Design; Simulation

ABSTRACT

Register transfer level (RTL) descriptions of digital systems have certain advantages over other descriptive techniques, especially during early phases of the design effort. There are at least three identifiable major uses for RTL-type descriptions. First, RTL can serve as documentation of digital processor behavior, recording in a concise fashion the operational characteristics of the system. RTL may also be used as the input notation accepted by an automatic translator which develops hardware structural details corresponding to the behavior described; output from such systems consist of appropriate logic modules, gates and other elements selected from a predefined library, along with suitable inter-connections. A third important application of RTL descriptions is in the simulation of digital systems, primarily during the system design process. In this case, RTL descriptions are processed by a portion of the simulation system, producing a model of the subject processor; as will be seen later, this model often includes structural as well as behavioral (control) elements. Initial conditions and external stimuli can then be applied to the model which, in conjunction with simulator facilities, produces appropriate outputs representing behavior of the simulated system. (Author)

(Technical Report No. CP 73013)

An Introduction to Register Transfer Level Simulation of Digital Systems

Cliff Hemming*
R. J. Smith, II

Department of Computer Science and Operations Research
Southern Methodist University
Dallas, Texas 75275

*Presently with Bell Telephone Laboratories, Columbus, Ohio.

Note: an expanded version of this paper will later appear as Chapter 3 of Volume II, Design Automation of Digital Systems, M.A. Breuer, ed.

May, 1973

Contents

Abstract

Register transfer level descriptions of digital systems have certain advantages over other descriptive techniques, especially during early phases of the design effort. There are at least three identifiable major uses for RTL-type descriptions. First, RTL can serve as documentation of digital processor behavior, recording in a concise fashion the operational characteristics of the system. RTL may also be used as the input notation accepted by an automatic translator which develops hardware structural details corresponding to the behavior described; output from such systems consist of appropriate logic modules, gates and other elements selected from a predefined library, along with .. table inter-connections.

A third important application of RTL descriptions is in the simulation of digital systems, primarily during the system design process. In this case, RTL descriptions are processed by a portion of the simulation system, producing a model of the subject processor; as will be seen later, this model often includes structural as well as behavioral (control) elements. Initial conditions and external stimulii can then be applied to the model which, in conjunction with simulator facilities, produces appropriate outputs representing behavior of the simulated system.

1. Introduction

Simulation using RTL descriptive techniques is attractive because it does not require detailed, comprehensive development of a proposed machine. In fact, an important use of RTL-based simulation is in evaluating alternatives during preliminary design of new systems. The machine architect thus has available facilities for observing behavior of proposed systems early in the design cycle. Issues which might be investigated using RTL simulation include:

a) execution speed of various instructions, instruction sets, or other gross timing studies,

b) the impact of unusual instructions or unconventional instruction implementations, such as memory search, move multiple character, translate, or out of line instruction execution,

c) hardware resource utilization, such as congestion and backup problems in pipelined organizations,

d) task switching and asynchronous interrupt handling capabilities,

e) for reconfigurable or fault-tolerant designs, evaluation of performance in various modes of operation.

One of the most interesting potential applications of RTL simulation is utilization of the simulator model as a vehicle for software development during the period before hardware is available. For the most part, this approach has been limited to comparison of capabilities of proposed and existing machines. Since RTL simulation models typically require several hundred to several thousand host machine instructions per (simulated) subject machine instruction, widespread use in software development hinges on availability of a very efficient RTL simulation system.

Register transfer level simulation is generally useful in evaluating and optimizing the architectural design of a digital system, rather than in uncovering races, hazards, illegal states and other critical timing considerations. The latter are normally considered in the domain of gate level simulation (see chapter 3 of volume 1). Gate level logic simulation requires detailed and quite comprehensive designs, while RTL simulation utilizes a more macroscopic, behavioral specification which is much more concise. Functional simulation, discussed in the next chapter, includes some properties of detailed simulation at the gate level while retaining much of the behavioral emphasis of RTL simulation.

Register transfer level descriptions have many attributes which make them highly desirable as simulation inputs. They are relatively concise, easily understood and are easily produced by system users; most importantly, they can take advantage of well known (programming) language translation techniques. Many of the translation approaches commonly employed in compilers, as well as those described in the last chapter, can thus be applied to simulator development.

While a number of Register Transfer Languages have been proposed, a surprisingly small number of these have been seen full implementation as simulator input languages. Limited implementations demonstrating the feasibility of a new language for simulation are more frequently developed. Consequently, existing RTL simulators are relatively crude in terms of their internal sophistication; techniques derived from programming language compilation predominate. However, these simulators have demonstrated the value of RTL as a effective design aid.

In section 2 we will develop an overview of typical RTL simulators, presenting a system overview, user interface considerations, implementation issues, and an exploration of important design and evaluation criteria for RTL simulation systems. Section 3 describes a representative simulation system and typical subject system representations. Simulation mechanisms and implementation considerations are treated in section 4; alternate simulation system approaches are treated in section 5, with a brief evaluation of RTL concluding this chapter.

2. Overview of an RTL Simulator

Structural designs of RTL simulators typically follow the general scheme outlined in figure 3.1

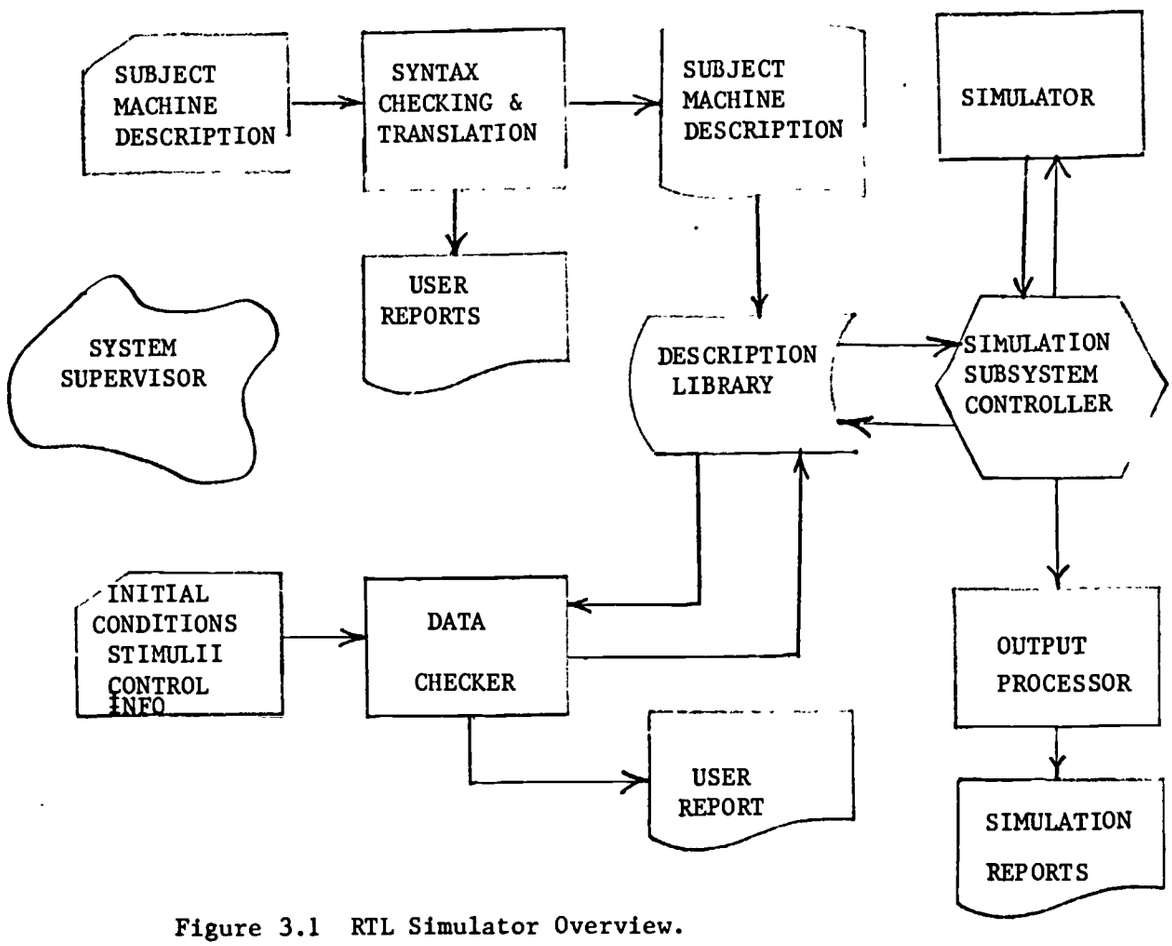

Figure 3.1 RTL Simulator Overview.

The subject machine RTL description is entered using an interactive or batch input device. The description is checked for valid syntax, and is translated into an internal format which is used to drive the simulation subsystem. In many practical systems, it has been found extremely useful to store the subject machine description on a "Description Library", as shown.

Simulation run control information, initial conditions and subject machine stimulii are processed by a separate data checker, which typically uses information stored in description library files. To facilitate use of the simulator system, input routines optionally produce a variety of printed reports summarizing processing accomplished on their inputs.

The simulation controller supervises actual simulation of the subject system; internal structure and function of the simulation subsystem is closely related to a number of issues detailed later in this chapter. The system supervisor is charged with invocation of appropriate subsystems, recovery from abnormal conditions, and other housekeeping tasks.

2.1 User Interfaces

One of the most important considerations in designing successful RTL simulation systems is the interaction of the system with end users. It should be possible for non-programmers to run simple simulations after minimal training. The system should be easy to use, with short but clear messages and diagnostics. Input conventions should be free form, with few hard-to-remember exceptions and tricks. (Experiences with

programming language compiler design can be easily applied to this aspect of RTL simulation.)

It has also been found useful for the syntax checker to perform a preliminary evaluation of the subject machine description, in an attempt to detect unintentional (but syntatically correct) specifications. These services might include: 1) Verification of data paths (which should always connect at least two elements), 2) Utilization of named quantities and conditions, 3) Evaluation of conditions imposed on operations (e.g., $(\overline{X} + X)$: $Y \leftarrow Z$), 4) Warning if unlikely specifications are input (e.g., 2000 bit wide bus), etc.

It should also be possible to update subject machine descriptions stored in the library, and to use alternate descriptions of functional subsystems, without reprocessing an entire system description.

Care should also be exercised in defining layouts for reports generated by the system. It is often necessary to add new features, optional outputs, or entirely new subsystems.

2.2 Implementation Issues

Internal representation of simulated network structure described in a register transfer language may take three forms: compiled code, tabular data or statements in a (source or internal) interpretative notation. Combinations of these representations might also be used.

Compiled code simulators require recompilation to incorporate

specification changes, and may require relatively large amounts of primary storage on the host machine. Performance is quite dependent on the sophistication of the translator. While most compiled code gate level simulators produce machine or assembly instruction output, RTL simulators of this variety are typically designed to generate a higher level source language output, e.g. PL/I, ALGOL or FORTRAN. The approaches differ because most gate level compiled code simulators have been delay free, and do not consider detailed timing relationships. RTL translators often use compiler techniques and consider complex timing situations, which are easier to implement using high level language facilities. The necessity for two translation operations prior to execution often leads to relatively inefficient performance; recently, the popularity of compiled code simulators has declined as table driven systems have demonstrated their superiority.

Tabular representation of subject machine structure appears to be more applicable to RTL simulation than compiled code techniques. Two simple tables are often used: the DEVICE TABLE might include entries specifying device type, specific attributes, input and output connections. Timing requirements could be kept in a second table. Note that timing characteristics are associated with operations involving several devices at the RTL simulation level, whereas timing specifications are typically associated with gate-level simulated devices.

Interpretation of source or intermediate code is another technique which has been used in RTL simulation. "Statements" describing subject

network structure are executed (interpreted) as they are encountered in the source language stream, generating results at each statement. With this technique, however, certain statements describing event or time-dependent processor actions may never be executed. Interpretation simulation systems must therefore include mechanisms (usually non-interpretative) for handling asynchronous conditional actions. Due in part to the relatively slow interpretative process, these techniques have enjoyed limited success in production environments.

As with many large systems, it is possible to trade memory requirements for system capabilities and performance in the design of an RTL simulator. It is therefore important to determine both the desired level of (simulated) detail and maximum capacity of the system before fixing the simulator design.

Simulator implementation strategy is a significant factor in determining servicability of the system. For example, concise representation of a modular subject system may be important to some users, while detailed analysis of flow of control in the subject network may be much more significant to another.

Implementation methodology may also place artifical constraints on the user; a weakness of some RTL simulators is the inability to accurately model asynchronous behavior.

Four of the most important system characteristics which should influence implementation are:

1) Treatment of asynchronous events, including external interrupts of the simulated system.

2) Facilities for describing simultaneous operations in the register transfer language.

3) The simulator's internal driving structure.

4) Techniques for generation and application of simulated stimulii.

2.3 Asynchronous Events

Asynchronous events constitute, in this context, the class of events whose time of occurrence is not known a priori. A number of processor functions are asynchronous; notable examples are "memory ready" on a read or write, and external interrupts. A memory ready may be treated as a known time event by setting a delay longer than the longest expected delay: this approach is not valid for processors that use mixed speed memory or interleaving; furthermore, this technique will not properly simulate the arrival of asynchronous interrupts. These events must be handled in simulation by a method which provides for detecting event occurance, and defining a technique for simulating the detected asynchronous event. Such facilities might resemble the PL/I user-defined ON-condition feature, which allows definition of a wide range of reponses to the occurance of asynchronous events.

Another method for achieving the required capability is to accumulate in tabular form all events which must be invoked asynchronously; this, in turn, requires that specification of asynchronous conditional events be explicitly identified. The construct "IF EVER conditional-expression THEN action-specification could be used for this purpose.

Then, prior to each simulation cycle, those conditional expressions having changed components must be evaluated to determine whether the asynchronous event has occured. This requirement can significantly increase the cost of a simulation, and should not be incorporated unless the additional overhead can be justified.

2.4 Simulation of Simultaneous Operations

The distinction should be made between asynchronous events, as discussed above, and asynchronous processes. An asynchronous process is one that proceeds independently in parallel with another (related) process. A common example is execution of a channel program concurrently with a central processor program. Successful simulation of such concurrency leads to the problem of simultaneous operations.

Simple simultaneous operations might be characterized by a register exchange. In the hardware, exchanging the contents of a pair of registers often requires no intermediate registers, and both registers are active simultaneously. This type of simultaniety is not difficult to simulate, although it usually must be simulated in most host computers by using an intermediate storage location (using a compiled simulation philosophy).

The more complex situation occurs when concurrent processes are active and accurate determination of the process which finishes last is critical, as with the channel and central processor. If these capabilities are required, they impose critical design requirements on the time flow mechanism of the simulator, since it must be capable of handling

multiple (possibly event driven) concurrent processes.

Timing mechanisms are usually based on fixed clock cycles or on extremely fine time accounting. Fundamental clock cycle (also called "fixed time increment") simulators are suitable for modeling completely synchronous systems, or when resolution of detailed timing situations is not important. This method of managing simulated time is relatively easy to implement and computationally efficient.

A more refined technique for handling timing of simulated events does not establish a fundmental period, but rather maintains a detailed resolution of event timing in the host computer. Using either technique, it has been found useful to use an event-driven driving mechanism. In order to avoid simulation of time periods when nothing changes, the simulator is invoked only at "times" when activity causes changes in the subject network. A time queue can be used to identify (simulated) times in the future when events are to occur.

2.5 Generation of Stimulii

Two objectives of gate level simulation are to detect timing defects and to develop system signatures of the network under various fault conditions. These are not usually objectives during register transfer level simulation. Emphasis is rather placed on such things as finding saturated and sparsely used data or control paths, developing statistics on element utilization, or improving machine throughput by balancing activities in the system. In gate level simulation, generating the complete

set of input vectors may be valid; in RTL simulation, the objective is to simulate classes of algorithms which stress various resources of the simulated machine. Automatic generation of such stimuli has received little publicity to date; it would appear that automated or semi-automatic generation of typical system inputs should be available to users of an RTL simulator. Implementation might involve definition of special input stimulus generator modules, or could be developed from libraries containing typical inputs having various characteristics. A straight forward, though perhaps non-trivial, method would utilize actual programs running on a current computer to generate the appropriate machine language stream for the simulated machine using a program specifically written for this purpose. This program could be modified to generate correct driving code for the simulated system as the configuration progressed. While such techniques have not found wide application, this situation may be attributed to a lack of wide spread use of RTL simulation techniques rather than some theoretical difficulty.

3. RTL Simulator Structure

A number of successful simulation lanaguages have been written using the Algol structure; to introduce RTI simulator mechanisms, this section investigates a specific RTL simulator form input, tracing proces ing through the internal mechanism, to actual simulation. Other simulators are briefly contrasted. A brief summary of Algol is presented to orient readers not familiar with that language.

3.1 Algol as a Programming Language

Algol is a high level language quite different from Fortran. It is a block structured language with dynamic storage allocation, having both local and global variables; expressions may be arithmetic, Boolean, and pointer in type, and result in values assigned to variables.

Iterative mechanisms much more powerful than the simple Fortran DO are available, and conditional or unconditional branches to alphabetic labels are allowed. Procedures are similar to subprograms but may have block structure.

Of particular interest at this point are 1) the block structure which may correspond to grouping of simulated physical components of the system; 2) declarations, which designate type (such as Boolean); and 3) conditional qualification of statements or expressions, which allows many types of logical tests to be specified.

3.2 The Computer Design Language Simulator

Computer Design Language is a RTL developed by Yaohan Chu (8,9) which has several Algol-like features. This section

presents concepts in RTL Simulator structure using CDL as an input language with other RTL features mentioned where appropriate.

Consider the network shown in figure 3.2, which consists of a serial shift register, and associated logic to form a complementor. The objective is to

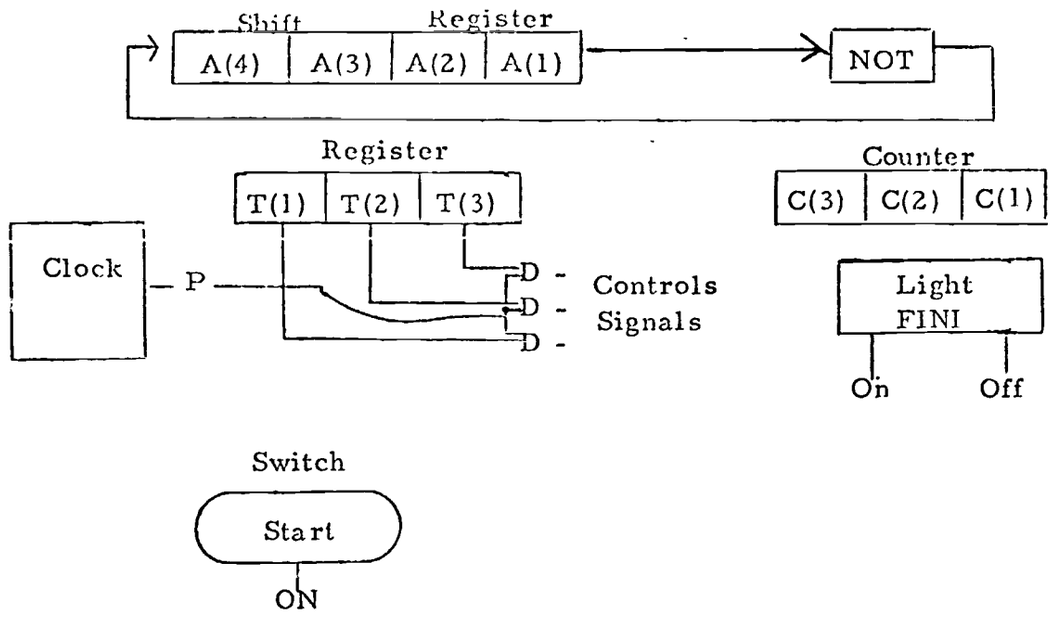

FIGURE 3.2 Complementor

model the behavior of the network at the RTL level. First consider the actual series of statements defining this network, and its action; later we will analyze these statements in detail.

```
1       $ TRANSLATE
2       * MAIN
3         Register, A (1-4),
4       1           T (1-3),
5       1           C (3-1),
6       1           FINI
7         Switch,   Start (On)
8         Clock,    P
9       /Start (On)/  T=4,
10                    FINI=O,
11                    C=O
12      /T(1)*P/      A(1-4)=A(4)[1] - A (1-3)
13                    C=C. Count.,
14                    T (1,2)=1
15      /T (2)*P/     IF (C. EQ.4) Then (T(2, 3) =1) Else (T (1, 2)=2)
16      /T (3)*P/     FINI = 1
17                    END
18      $ SIMULATE
19      * OUTPUT CLOCK (1) = A, T, C, FINI
20      * SWITCH  1, Start = On
21      * LOAD
22                    A=16
23      * SIM       30, 3
24      * RESET     CYCLE
25      * LOAD
26                    A=05
27      * SIM       30, 3
28      1
```

FIGURE 3.3 CDL Description of Complementor

First, observe that the RTL simulator input is divided into two sections, TRANSLATE & SIMULATE (each preceded by the $). CDL requires only these two input sections; the first translates the source language into a form (polish string) which the simulator can interpret. The second section controls simulation of the subject system description based on the string.

The TRANSLATE section provides a description of the system to be simulated and its desired behavior to the system. The first few statements are DECLARATIONS, specifying component attributes: the registers, clock, and the switch. Note that the light FINI is declared as a register. The device types available in CDL are REGISTERS, SUB-REGISTERS, MEMORIES, DECODERS, SWITCHES, TERMINALS (output of elements without storage which manipulate data such as an adder), BLOCKS (parallel interaction between the devices), and CLOCKS.

The remaining statements of the TRANSLATE section form the LABELED STATEMENT section. Each is composed of a LABEL followed by one or more MICRO-STATEMENTS. Labels consist of logical expressions, with or without a clock, which are evaluated and are true or false. If true, the associated micro-statements are performed. Any number of labels may be true at once, implying parallel operation.

Micro-statements determine the functioning of the digital system. They allow logical expressions to be formed and the result (of 1 or more bits) to be assigned to a storage element. Micro-statements may be simple (unconditional) or conditional, corresponding to the Algol IF statement syntax.

The SIMULATE section invokes the simulator routine of the CDL system to initiate the simulation process. In this example, the simulator

runs a maximum of 30 cycles (in this case, fixed time increment clock cycles) with the restriction that the same group of labels may not be true more than 3 consecutive intervals (*SIM 30, 3).

Keeping the above example in mind, let us explore details of the simulation process. First, consider the action of the network as defined by the components and microsequences, and the manner in which a simulator might model the subject network.

The complementor is set to its initial value of $16_8=1110_2$ (Reference line 22) in the SIMULATE section. When the switch is turned on, the T register is set to 100_2, and C goes to 000; thus T (1)=1 and at clock period 1 the statement A (1-4) = A(4)^1-A (1-3) are "executed", effectively generating a right circular shift with complement. Note that the right side is completely evaluated and saved until the end of the clock sycle; more on this later. The reader may verify that following outputs produced by the simulator are as shown in Figure 3.4, where the system initially starts with A=5 as shown in Line 26.

CLOCK TIME	LABEL	A 1 2 3 4	T 1 2 3	C 3 2 1	FINI
0	/SWITCH(ON)/	0 1 0 1	1 0 0	0 0 0	0
1	/T(1)* P/	0 0 1 0	0 1 0	0 0 1	0
2	/T(2)* P/	0 0 1 0	1 0 0	0 0 1	0
3	/T(1)* P/	1 0 0 1	0 1 0	0 1 0	0
4	/T(2)* P/	1 0 0 1	1 0 0	0 1 0	0
5	/T(1)* P/	0 1 0 0	0 1 0	0 1 1	0
6	/T(2)* P/	0 1 0 0	1 0 0	0 1 1	0
7	/T(1)*P/	1 0 1 0	0 1 0	1 0 0	0
8	/T(2)* P/	1 0 1 0	0 0 1	1 0 0	0
9	/T (3)* P/	1 0 1 0	0 0 1	1 0 0	1
10	/T(3)* P/	1 0 1 0	0 0 1	1 0 0	1
11	/T (3)* P/	1 0 1 0	0 0 1	1 0 0	1

END

FIGURE 3.4 Typical values produced by CDL simulation of the complementor

Now let us summarize the concepts important from a user point of view in the RTL simulation.

(1) The circuit must be described, conveniently, at the register level.

(2) The action of the circuit must be described, and in particular, provision for simultaneous activity should be included.

(3) The system must be initialized.

(4) Simulation should occur until either a pre-specified time has elapsed, a specific event occurs, or a certain condition applies

(5) The user should be able to specify desired forms of output data.

(6) Several simulation passes should be available in one run.

These, and other features are provided by CDL. Of course, similar features may be provided in other ways. Such considerations are discussed in section 4, and other approaches are presented in section 5. Now, let us look at the internal features of our example CDL simulator.

3.3 Internal Features of a CDL Simulator.

The underlying structure which supports simulator activity determines the speed, accuracy, capacity, and flexibility of the system.

This section is based on features of the internal structure of CDL (Version 2) (22). The two portions, TRANSLATE and SIMULATE, are closely intertwined, since TRANSLATE builds tables used by SIMULATE.

TRANSLATE accepts input describing the system and generates tables reflecting the system's structure. The tables for the example

system are:

(a) Subprogram,

(b) Label,

(c) Switch Label,

(d) Clock

(e) Symbol (declaration names)

(f) Storage Array

The Subprogram Table contains entries which associate entries in other tables with each specific subprogram. Each subprogram (including Main) has a set of 7 entries. These entries consist of 1) the subprogram name, 2) and 3) first and last entries, for this subprogram, in the Label Table, 4) and 5) first and last entries in the Switch Label Table, 6) pointer to the polish strings (see below) and 7) index in the Symbol Table.

The Label Table has two entries: 1) pointer to the polish string for this label; and 2) the label name.

The Switch Label Table has 3 entries: 1) The switch name; 2) the switch position; 3) polish string pointer.

The Clock Table has 4 entries: 1) The clock name, 2) Number of clock, 3) pointer to the next occurring clock time and 4) count of the number of elapsed clock cycles.

The Symbol Table contains information about the devices formed by the Declarations. There are entries for each device type declared, including the device type, number of simulated bits, bit ordering and index, names and related information.

Finally, the Storage Array Table is a dynamic area used by the SIMULATE routine to store the intermediate results (temporary results generated during a cycle) as well as the permanent results at the cycle end. Each device requiring storage has a permanent entry assigned for the duration of the simulation.

The TRANSLATE section also produces a polish string for each expression requiring evaluation at simulation time, including micro-statements, labels, terminals and decoders. The strings are divided into segments. A segment consists of either 1) a label expression, 2) a terminal expression, 3) a block of micro-statements, or 4) all micro-statements associated with a specific label. The polish strings of course, have an area of memory reserved for their storage.

After processing of the TRANSLATE sections is completed, the SIMULATE section is invoked. Five important component programs here are:

(a) Loader

(b) Output processor

(c) Switch

(d) Simulate

(e) Reset

The loader initializes the simulated digital system to a desired initial state by reading values from input data and (effectively) storing these values for declared devices in the Storage Array Table.

The OUTPUT program prints the values associated with various devices, such as registers, switches, or memory, at various user selected intervals.

The SWITCH program allows the user to set switches (equivalent to manual setting of a physical switch) at a specific time.

The SIMULATE routines interpret micro-statements, generating results for each SIMULATED cycle. The routine orders processing so that 1) statements associated with true switch labels are executed (only the first time the switch becomes true), 2) Labels are evaluated, 3) all micro-statements associated with true labels are evaluated, 4) Evaluated results are stored (after all statements are evaluated), 5) The next cycle is performed or simulation is terminated.

The RESET program resets, at the user's discretion, the clocks, output, switches, or cycle counter, either separately or in combination.

Of these routines, the workhorse is the SIMULATE program; thus it is the most critical to performance of the simulator (but not necessarily to user acceptance of the system as a useful tool).

4. Simulator Mechanisms and Implementation Considerations

There are several major aspects of register level simulation that need to be emphasized. These are:

1. Ease of description of the simulated digital process.
2. Accuracy in controlling the continuing simuation
3. Fast and economical implementation of the simulator (good structure)
4. Control of output

These will be discussed in turn.

The ability to easily and accurately describe a system, and control its flow might best be described by the pathological cases, for example, many computers have an Exchange instruction i.e., A $\longleftrightarrow$ B. How is this accurately modeled? Certainly not by the exchange instruction of the host computer (if any), since the word length of the simulated machine may be longer than that of the host. Consequently, the technique of evaluating all expressions, generating results in a separate special place (not declared in the simulated machine) and then, after all evaluations are complete, storing away the result, solves the problem; this, however, impacts the complete philosophy of simulation. The net result of using this convention is that a change produced during the clock interval cannot be used until the next clock interval, occasionally requiring the user to consider quite carefully the way the system is described to the simulator.

Thus $C = A + B$

$F = C + D$

would not use the new value of C; these would require separate time

periods (LABELS in CDL). This simple example illustrates a relatively complex problem: implementation of RTL simulator timing mechanisms is an extremely critical part of simulator design.

To pursue timing further, one may attempt to parallel gate level simulator timing mechanization (See Vol. 1, Sect. 3.3.2) in designing timing mechanisms. At the gate level, however, the concept of a gate delay is quite concrete and very realistic entity. On the other hand, in a register level model the same machine may require critical timing of either processes (a series of transfers), or individual transfers, or both. The consequence of these considerations is that the underlying simulator mechanism must support the fine detail timing, and defaults should be assumed to alleviate the necessity for describing each timing dependency by the user. The fine detail requires that zero delay (the exchange instruction mentioned previously) be considered simultaneously with timed processes. The implementation of these combined capabilities has been found to require relatively large simulation overhead.

A second critical feature in the implementation is the type of functions available to the user and how these are defined. It is reasonable to have an "add" operation for instance. However, is it 1's complement, 2's complement or what exactly? This is another critical issue, and it is an important design decision. The simulator should provide a well-defined standard and a method to alter the default. As another example, consider a branch on zero instruction --- what is zero (+0,-0, or both), how is it represented), and where is the check to be made? As the accuracy and flexibility of the models provided increases, the RTL simulator design begins to resemble a gate level

simulator in underlying complexity and simulation over head, thereby increasing the detail required of the user. This trade off is a very complex issue.

Let us consider now an implementation strategy based on a structure which 1) uses tables to store all data, including device type, memory, interconnect, and timing information, and 2) an interpretative technique for controlling executing of the simulation. This method provides a highly flexibile simulator with simple descriptive input at a cost of storage space and speed of simulation, a trade off we believe most advantageous in view of current trends in host machine configurations. The table structure permits handling many diverse devices, unlimited combinations of register and data path widths, and complex timing constraints in a straight forward manner. The interpretation of control flow allows quite succinct control descriptions by the user, allowing recursive constructions to be implemented by well known techniques.

As previously noted, simulator output is the user's view of the simulated system behavior. Output should easily and rapidly convey the system behavior in terms the user can readily grasp. Often ignored or considered lightly, the presentation or format of the output, and user ability to control the output, can lead to failure of an otherwise efficient and carefully designed system. In particular, the output should be event oriented; i. e., only changes in system state, and indeed only those specified, should generate output unless the user specifies otherwise (which he should be able to do). The import of carefully designed reporting routines is difficult to overemphasize, and becomes quite critical if the system is to be used in an interactive simulation environment.

5. Alternate System Approaches

There are many ways to realize a model capable of supporting a system simulation. In addition to DDL discussed in chapter 2 and CDL discussed here, simulators have been described for most RTL's discussed in chapter 2. In this section, we briefly describe several of these systems to develop some contrast to the CDL approach presented earlier.

5.1 Compiled Simulation

A compiled simulator generates machine instructions that represent the action of the machine; these are then executed for various initial values of the simulated machine. An example of a compiled description may be obtained by writing a Fortran description of the system. A Fortran program which models the system shown in Figure 3.2 is shown in Figure 3.5.

```
*FT TEST.F4
      LOGICAL RA(4),PT(3),FINI,SW,CLOCK
      LOGICAL RAA(4),RTT(3),FINII,SSW,CCL
      INTEGER RC,RCC,CYCLES,CHGS
      LABELC=0
      NCHG=0
      CLOCK=.TRUE.
      SSW=.FALSE.
      CCL=CLOCK
      ACCEPT 1,SW
      ACCEPT 3,RA
      ACCEPT 2,CYCLES,CHGS
      IF(SW.NE.SSW)GO TO 100
5     IF(RTT(1).AND.CCL)GO TO 200
15    IF(RTT(2).AND.CCL)GO TO 300
25    IF(RTT(3).AND.CCL)FINI=.TRUE.
      CYCLES=CYCLES-1
      IF(CYCLES.EQ.0)GO TO 999
      DO 30 I=1,4
      IF(RAA(I).NE.RA(I))NCHG=0
30    RAA(I)=RA(I)
      DO 40 I=1,3
      IF((RTT(I).NE.RT(I)).OR.(RCC.NE.RC))NCHG=0
40    RTT(I)=RT(I)
      RCC=RC
      IF((FINII.NE.FINI).OR.(SSW.NE.SW))NCHG=0
      FINII=FINI
      CLOCK=CCL
      NCHG=NCHG+1
      SSW=SW
      LABELC=LABELC+1
      TYPE 6,LABELC
      TYPE 4,RA,RT,RC,FINI,CLOCK,CYCLES
      IF(NCHG .EQ. CHGS)GO TO 999
      GO TO 5
100   RT(1)=.TRUE.
      FINI=.FALSE.
      RC=0
      GO TO 5
200   RA(1)=.NOT.RAA(4)
      RA(2)=RAA(1)
      RA(3)=RAA(2)
      RA(4)=RAA(3)
      RC=RCC+1
      RT(1)=.FALSE.
      RT(2)=.TRUE.
      GO TO 15
300   IF(RC.EQ.4) GO TO 301
      RT(1)=.TRUE.
      RT(2)=.FALSE.
      GO TO 302
301   RT(2)=.FALSE.
      RT(3)=.TRUE.
302   GO TO 25
999   CALL EXIT
1     FORMAT(L1)
3     FORMAT(4L1)
2     FORMAT(I2)
4     FORMAT('  A ',4L1,'  T ',3L1,'  C ',04,'  FINI ',L1,'  CLOCK ',
     X L1,' CYCLES',I5)
6     FORMAT(' LABEL CYCLE',I4)
      END
```

Figure 3.5 Compiled Simulation for the Example Complementor

```
.FX TFST.F4
LOADING

LOADER 2K CORE
EXECUTION
T
FTFT
30
3

LABEL CYCLE   1
 A FTFT  T 1FF  C 0000  FINI F  CLOCK 1 CYCLES   29
LABEL CYCLE   2
 A FFTF  T FTF  C 0001  FINI F  CLOCK T CYCLES   28
LABEL CYCLE   3
 A FFTF  1 TFF  C 0001  FINI F  CLOCK T CYCLES   27
LABEL CYCLE   4
 A TFFT  T FTF  C 0002  FINI F  CLOCK T CYCLES   26
LABEL CYCLE   5
 A TFFT  T TFF  C 0002  FINI F  CLOCK T CYCLES   25
LABEL CYCLE   6
 A F1FF  T FTF  C 0003  FINI F  CLOCK 1 CYCLES   24
LABEL CYCLE   7
 A FTFF  T TFF  C 0003  FINI F  CLOCK 1 CYCLES   23
LABEL CYCLE   8
 A TFTF  1 FTF  C 0004  FINI F  CLOCK T CYCLES   22
LABEL CYCLE   9
 A TFTF  T FF1  C 0004  FINI F  CLOCK T CYCLES   21
LABEL CYCLE  10
 A TFTF  T FFT  C 0004  FINI 1  CLOCK T CYCLES   20
LABEL CYCLE  11
 A TFTF  T FFT  C 0004  FINI T  CLOCK T CYCLES   19
LABEL CYCLE  12
 A TFTF  T FFT  C 0004  FINI T  CLOCK 1 CYCLES   18

CPU TIME: 0.40  ELAPSED TIME: 1:33.67
NO EXECUTION ERRORS DETECTED

EXIT

 EX TEST.F4
LOADING

LOADER 2K CORE
EXECUTION
T
TTTT
30
3

LABEL CYCLE   1
 A TTTT  T TFF  C 0000  FINI F  CLOCK T CYCLES   29
LABEL CYCLE   2
 A FTTT  T FTF  C 0001  FINI F  CLOCK T CYCLES   28
LABEL CYCLE   3
 A FTTT  T TFF  C 0001  FINI F  CLOCK T CYCLES   27
LABEL CYCLE   4
 A FFTT  T FTF  C 0002  FINI F  CLOCK T CYCLES   26
LABEL CYCLE   5
 A FFTT  T TFF  C 0002  FINI F  CLOCK T CYCLES   25
LABEL CYCLE   6
 A FFFT  T FTF  C 0003  FINI F  CLOCK T CYCLES   24
LABEL CYCLE   7
 A FFFT  T TFF  C 0003  FINI F  CLOCK T CYCLES   23
LABEL CYCLE   8
 A FFFF  T FTF  C 0004  FINI F  CLOCK T CYCLES   22
LABEL CYCLE   9
 A FFFF  T FFT  C 0004  FINI F  CLOCK T CYCLES   21
LABEL CYCLE  10
 A FFFF  T FFT  C 0004  FINI T  CLOCK T CYCLES   20
LABEL CYCLE  11
 A FFFF  T FFT  C 0004  FINI T  CLOCK T CYCLES   19
LABEL CYCLE  12
 A FFFF  T FFT  C 0004  FINI T  CLOCK 1 CYCLES   18

CPU TIME: 0.43  ELAPSED TIME: 1:33.57
NO EXECUTION ERRORS DETECTED

EXIT
```

Figure 3.6 Compiled Simulation Output

As you may observe, this program is considerably longer, and the registers (RC and RCC) are simulated as integers instead of registers. Nevertheless, it would accurately model the complementor, although very inefficiently.

5.2 Other Existing RTL Simulators

Several of the RTL simulators described earlier have been implemented. In this section several of these are briefly reviewed.

DDLSIM, A Digital Design Language Simulator [1] is a simulator for DDL. Darringer [12] describes a simulator which accepts APDL, Algorithmic Processor Description Language. SODAS has been partially simulated in ALGOL and BOOLE [26]. Both APDL and SODAS use Algol-like expressions as the network description source language; simulation is performed in a dialect of Algol. DDLSIM uses DDL (see previous chapter) as its descriptive input, and the simulation is performed by a group of 8 FORTRAN programs. APL is a general purpose programming language which has been used to describe processors; interpretation of APL simulation input has been described using assembly language routines developed by IBM.

5.2.1 Simulators Using ALGOL-Like RTL

Register transfer languages resembling the ALGOL programming language are popular, since from a simulation viewpoint, network description expressed in this fashion are particularly attractive:

1. Descriptions of sub-elements may be made modular, corresponding to ALGOL compound statements.
2. System inputs and outputs are conveniently identified.
3. Subsystem definitions may be independently prepared,

and these in turn may be combined to produce a system definition.

4. Structural (component and interconnection) descriptions may be combined with behavior specifications.
5. Subsystems may be described and hence simulated at a level of detail appropriate to each subsystem.

Asynchronous events must be defined using the APDL "if ever" statement, which is actually a declaration of a sequential process initiated whenever the specified conditions are satisfied. Each such process (defined by the "body" of an "if ever" statement) must not be reactivated until it terminates and requires at least one basic cycle time unit.

5.2.2 LOTIS

Lotis (Logic Timing Sequencing) is a comprehensive hardware notational language suitable for simulation [27]. It embodies certain aspects of both Algol and APL, but is quite distinct from either. To perform a simulation, the language is extended to provide initiation methodology and describe primary statistic gathering and the analysis requested.

The body of a description in Lotis consists of 2 parts: a declarative part and a procedural part. The division which generates the structure, timing, and logical properties to be simulated is the declaration; the machine actions to be simulated are described in the procedure. In the Algol sense, the procedural portion corresponds to the body of a block

with its declarations.

Lotis has many features which allow complex timing to be accurately described. Specific delays may be associated with the various operators; sections within the procedure may be "interlocked" to establish dependencies on other sections; and concurrency may be present in different control sections. Further, irrespective of delays associated with an operator, a particular transfer may be assigned a specific delay. This delay may in turn be combined with operator delays.

The procedural portion is composed of a number of entities which represent the various autonomous control mechanisms of the simulated machine. These are called groups. For example, the memory may constitute a group, with separate divisions for the read access and the write initiate. A group is composed of functions or sequences or both, and the timing interlocks may be associated with the group.

Both functions and sequences describe the logical action of a functional unit of the process, and are composed of a series of steps. The primary difference is that a function yields answers without exhibiting structured detail in the code, while a sequence describes the intimate details of a process, such as all the bit transfers inherent in stepping a counter.

Within a sequence the steps are normally executed in order, although the sequence may be entered at any point. Three distinct time relations may exist between the steps: they may be asynchronous, fixed delay, or synchronous.

In the asynchronous case, the step interval is determined by the delay of operators in the statement. Fixed Delay timed steps have an explicit delay associated with the step. Synchronous, in the Lotis case, means the step is interlocked or conditioned on a variable (which supposedly is a clock). These timing features may be combined.

These features along with branch control, conditional assignment, global assignments (similar to Fortran statement functions) and other features make Lotis a powerful description language well suited to accurate RTL simulation.

5.2.3 DDLSIM [1]

DDLSIM is a simulator for the Digital Design Language discussed in the preceding chapter. DDLSIM is a Fortran system, consisting of 9 programs used in two phases. The first phase accepts the source description and compiles the executable instruction string. The second phase schedules the strings for execution by an interpretive processor.

The simulator is essentially a unit time increment simulator, completely evaluating a state prior to advancing to the next clock interval. The Scheduler program is the heart of the system, continually examing the timing tables to determine the next state for the simulated machine. In the DDLSIM context, a machine is analogous to functional module such as memory, arithmetic unit, channel, etc. Variable timing may be associated with each module, but must be expressed in the context of unit time steps.

On conclusion of a DDLSIM run, statistics are available to the designer, including registers undergoing change, with a trace and all altered registers.

5.3 Interactive Systems

Until recently, RTL simulation was available only in batch processing environments. As a consequence, simulation was at best an awkward design aid in early phases of system design specification. However, it has recently become clear that the most effective use of RTL simulation is early in the development cycle, when information requirements most closely match the capabilities of RTL simulation.

Interactive simulation allows the designer to study and experiment with design alternatives during initial, creative phases of the system development. Modification of system descriptions, and evaluation of proposed design behavior can be accomplished rapidly using RTL interactive simulation support.

6. Summary

It is apparent that no register transfer language has attracted the following that some of the general purpose simulation languages such as GPSS or SIMSCRIPT have enjoyed. Indeed, a number of recent simulators of digital systems have been written in general purpose programming language such as Fortran (the PDP-11, for example), sidestepping not only RTL's but general purpose simulators as well. The trend is not new, and its existence is well documented. [23]

A number of reasons exist for this failure to utilize special simulators. At present none of the major computer manufacturers support one of the systems as a part of its distributed (free or otherwise) software. Easily identifiable problems are that these existing systems are not readily available, not well documented, cannot accurately model the various special components available, or are so general that the cost of using such a system can be prohibitive. This is not to indicate that some

industrial firms do not regularly use an internal simulator, but that such simulators are usually well tuned to the particular equipment that such manufacturers produce.

However, it may well be that the time is near when register transfer simulator will meet with success. The PMS (processor-memory-switch) and ISP (instruction set processor) notations developed by Bell and Newell [5] and used in that text to describe a number of systems, indicate the broad applicability of these notations. Work on simulators for these is progressing [20]. With the advent of popular MSI from which processors are currently being constructed, an increased standardization may be expected in components used in design. Simulation and simulators may soon become more straightforward and allow concentration on the development of register level models representing efficient, effective solutions meeting design objectives.

This approach may be enhanced by the development of functional level digital simulation. Work in this area is relatively new [8, 10, 16, 31], but has demonstrated the potential for combining into one system the advantages of RTL and gate level simulation. In the past, the objective for RTL and gate level simulation have been somewhat different, as discussed earlier in this chapter. However, functional level simulation seems to have a number of characteristics common to both. Due to the infancy of functional simulation, it would be premature to consider its effect on RTL simulation, but it is clear that functional simulation is adding a new dimension to the area of digital logic simulation.

References

1. Arndt, R. L. and Dietmeyer, D. L., "DDLSIM"--A Digital Design Language Simulator," Proceedings of N.E.C., Vol. 26, Dec. 1970, pp. 116-118.

2. Baray, M. B. and Su, S. Y. H., "A Digital System Modeling Philosophy and Design Language," Proc. Eighth Design Automation Workshop, June 1971, pp. 1-22.

3. Baray, M. B., et.al., "The Structure and Operation of a System Modeling Language Compatible Simulator," Proc. Eighth Design Automation Workshop, June 1971, pp. 23-24

4. Baray, M. B., et. al., "A System Modeling Language Translator," Proc. Eighth Design Automation Workshop, June 1971, pp. 35-39.

5. Bell, G. C., and Newell, A., Computer Structures, Readings, And Examples, New York: McGraw-Hill, 1971.

6. Breuer, M. A., "Functional Partitioning and Simulation of Digital Circuits," IEEE TC, Vol. C-19, No. 11, November 1970, pp. 1038-1046.

7. Change, H. Y. and Manning, E. G., "Functional Techniques for Efficient Digital Fault Simulation," Digest of the First IEEE Computer Group Conference (1967) pp.

8. Chu, Y., "An ALGOL-Like Computer Design Language," CACM, Vol. 8, No. 10, October 1965, pp. 607-615.

9. _____, "Design Automation by the Computer Design Language," Technical Report 69-86, Computer Science Center, University of Maryland, March 1969.

10. Cohen. D. J., "Computer Based Fault Analysis of Digital Systems." Research Report CSRR2020, University of Waterloo, Dept. of Applied Analysis and Computer Science, Waterloo, Canada, 1970.

11. Crall, R. F., "ICCAP--Interactive Computer Assistance for Creative Design," Ph.D. dissertation, University of Missouri, Rolla, 1970.

12. Darringer, J. A., "The Description, Simulation, and Automatic Implementation of Digital Computer Processors," Ph.D. dissertation, Carnegie-Mellon University, May 1969. (Also available as NTIS #AD 700144).

13. Duley, J. R. and Dietmeyer, D. L., "A Digital System Design Language (DDL)," IEEE TC, Vol. C-17, No. 9, September 1968, pp. 850-861.

14. Duley, J. R. and Dietmeyer, D. L., "Translation of a DDL Digital System Specification to Boolean Equations," IEEE TC, Vol. TC-18, No. 4, April 1969, pp. 305-313.

15. Friedman, T. D. and Yang, Sih-Chin, "Methods Used in an Automatic Logic Design Generator "ALERT)," IEEE TC, Vol. 18, No. 7, July 1969, pp. 593-614.

16. Hemming, C. and Szygenda, S. A., "Modular Requirements for Digital Logic Simulation at a Predefined Function Level," Proceedings of the ACM Annual Conference, August 1972, pp.

17. Iverson, K. E., A Programming Language, New York: Wiley, 1962.

18. Knuth, D. E. and McNeley, J. L., "SOL-A Symbolic Language for General Purpose Simulation," IEEE TEC, Vol. No. 13, August 1964, pp. 401-408.

19. _____, "A Formal Definition of SOL," IEEE TEC, Vol. No. 13, August 1964, 409-414.

20. Knudson, private communication.

21. McClure, R. M., "A Design Language for Simulating Digital Systems," JACM, Bol. 12, No.1, January 1965, pp. 14-22.

22. Mesztenyi, C. K., "Computer Design Language Simulation and Boolean Translation," Technical Report 68-72, Computer Science Center, University of Maryland, June 1968.

23. Nielsen, N.P., "Computer Simulation of Computer System Performance," Proc. ACM National Conference, 1967, pp.

24. Parnas, D. L., "A Language for Describing the Functions of Synchronous Systems," CACM, Vol. 9, No. 2, February 1966, pp. 72-76.

25. Parnas, D. L., "Sequential Equivalents of Parallel Processors," Computer Science Department, Carnegie-Mellon University, February 1967.

26. Parnas, D. L. and Darringer, J. A., "SODAS and a Methodology for System Design," Proc. FJCC, 1967, pp. 449-474.

27. Schlaeppi, H. P., "A Formal Language for Describing Machine Logic, Timing, and Sequences ILOTIS)," IEEE TEC, Vol EC-13, No. 8, August 1964.

28. Schorr, H., "Computer-aided Digital System Design and Analysis Using a Register Transfer Language," IEEE TEC, Vol. 13, No. 13, December 1964, pp. 730-737.

29. Szygenda, S. A., TEGAS2 -- Anatomy of a General Purpose Test Generation and Simulation System for Digital Logic," Proceedings of the 1972 Design Automation Workshop, June 1972, pp.

30. Stabler, E. P., "System Description Language," IEEE TC, Vol. C-19, No. 12, December 1970, pp. 1160-1173.

31. Szygenda, S. A., et. al., "Functional Simulation, A Basis for a Systems Approach to Digital Simulation and Fault Diagnosis," Proc. Annual Summer Simulation Conference, June 1972, pp.

32. Zuker, M. S., "LOCS -- An EDP Machine Logic and Control Simulator," IEEE TEC, Vol, EC-14, No. 6, June 1965, pp. 403-416.